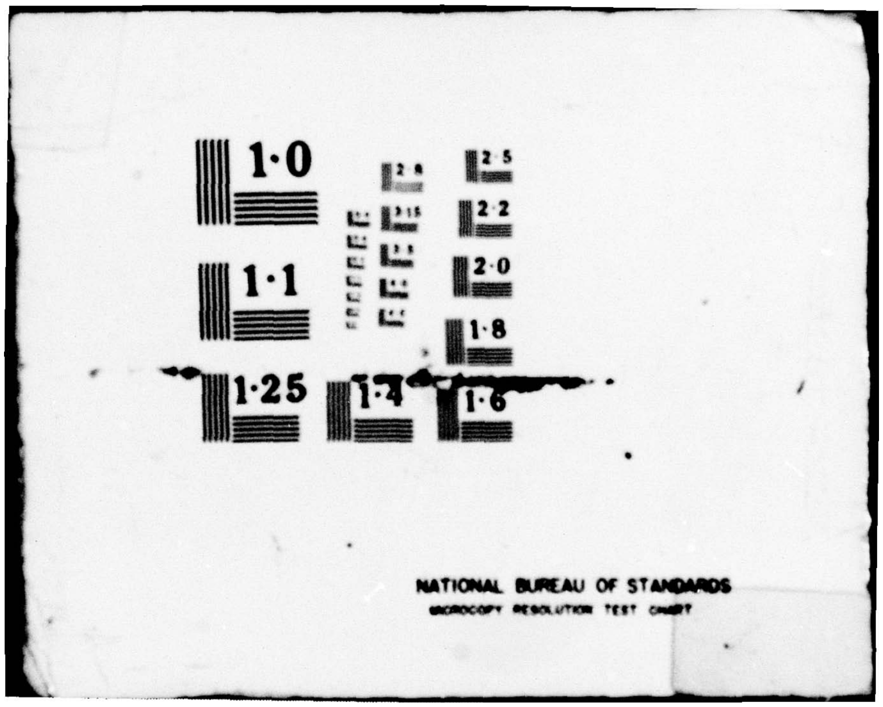
AD-A077 427 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 5/2
THE STRUCTURE OF SPECIFICATIONS AND IMPLEMENTATIONS OF DATA ABS--ETC(U)
SEP 79 M A ARDIS , R G HAMLET AFOSR-77-3181
UNCLASSIFIED TR-801 AFOSR-TR-79-1154 NL

1 OF 1
AD-
A077427

END
DATE
FILMED
1-80
DDC


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

TR-801 11 ~~September~~ 1979

The Structure of Specifications and Implementations of Data Abstractions

10 Mark A. Ardis
Richard G. Hamlet

12

Technical rept.

16

COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

(14) TR-801 (11) Sep~~tember~~ 1979

(6) The Structure of Specifications and Implementations of Data Abstractions,

(10) Mark A. Ardis
Richard G. Hamlet

(12) 51

(9) Technical rept.,

(16) 2304

Department of Computer Science
University of Maryland
College Park, Maryland 20742

(17) A2

(18) AFOSR

(19) TR-79-1154

The work reported here was supported by a grant from the Air Force Office of Scientific Research ~~(AFOSR3181B)~~. Computer time was provided in part by the Computer Science Center of the University of Maryland.

(15) AFOSR-77-3181

409 022

JOB

ABSTRACT --

A data abstraction is a collection of sets together with a collection of functions. An intuitive abstraction is unconnected with formalism: the sets and functions are supposed to be known ab initio. Formal ideas enter when the abstraction is (i) implemented, a conventional program written to carry out the operations on actual data; and (ii) specified, a mathematical characterization given to precisely describe its sets and functions. The intuitive abstraction, an implementation, and a specification share a syntax that names the sets and functions, and gives the function domains and ranges (as set names). The central question for any particular example of syntax is whether the semantics of the three ideas correspond: does the collection of objects and operations a human being was thinking of behave in the way the implementation's data and procedures behave? Do the mathematical entities behave as imagined? The questions can never be answered precisely, because the intuitive abstraction is imprecise. On the other hand, precise comparison of specification and implementation is possible.

This paper presents an algebraic comparison of specifications with implementations. It is shown that these abstractions always overlap, and have a common (lattice) structure that is valuable in understanding the modification of code or specification. However, in dealing with the precise entities subject to formal analysis, we must not lose sight of the intuition behind them. Therefore, our definitions are framed in terms of the intuitive abstraction a person attempted to specify or implement, and we refer the algebraic ideas to this standard whenever possible.

Section 1 presents the intuitive ideas of an abstraction, its implementation, and specification. The ideas are essentially those of [Hoare 72] and [Guttag 77]. Section 2 gives the common formalism to be used, the constant word algebra. In Sections 3 and 4, this is applied to specification and implementation. Section 5 explores the overlap between the ideas, and suggests that the precise connection can shed light on the imprecise one that is really of interest: the intuitive abstraction in a person's mind.

1. Data Abstractions

A data abstraction is viewed in three distinct ways in this paper. First, as an intuitive object, an abstraction is a collection of sets and mappings among them, with the sets and mappings having an intuitive existence in the mind of a human being. (Perhaps this existence is God-given, as Kronecker claimed of the natural numbers; perhaps it is man's handiwork; from whatever source we have it whole.) Second, an abstraction is what a programming language supporting type-encapsulation delivers. In the SIMULA CLASS [Dahl et al. 70], the CLU cluster [Liskov et al. 77], and the SIMPL-D CLASS [Gannon & Rosenberg 78], these languages provide the programmer with a means of implementing what he had in mind. But once written, code has an existence of its own, which need bear no relation to the ideas that remain in the person's mind. And third, an abstraction may be given a formal, mathematical definition. Indeed, one branch of mathematics is devoted to giving formal shape to intuitive ideas like numbers and sets. As anyone who constructs formalisms knows, they also acquire a life of their own, and the correspondence with intuition is a difficult one to establish.

The ideas of implementation and specification of an abstraction can be made formal and precise. As formal notions, they are independent of each other, and both are imperfect mirrors of an intuitive idea. The fundamental definitions of this paper involve <u>correctness</u> -- the precise statement of what it means for a formal implementation or specification to agree with intuition. Such definitions necessarily retain a nonmechanical portion, but it is important to know, before we pass to analysis of the formal idea, exactly what it is supposed to be standing in for.

The three ideas of abstraction share a syntax that names the collection of sets, names the functions, and gives the set names for function domains and ranges. The definition of this <u>signature</u> (Section 2) is unusual in that it includes only the names of the sets and functions, not those objects themselves. Most definitions of computer science -- the legendary 17-tuple of automata theory for example -- involve sets themselves. Here this is inappropriate, because to give a set is to give a meaning. The signature consists of names; when objects are attached to those names, it is the definition of the abstraction itself. If the assignment to the names arises from code, we have the implemented abstraction; if it arises from some mathematical description, the specified abstraction; and behind it all are the sets and functions of the intuitive abstraction. Much of the power of formal ideas comes from a convenient confusion of syntax with semantic substance: we talk about the complete entity as if it had only syntax. This confusion is too expensive here, because with three possible meanings, we can never explicate the relationships among them from their common names.

It is impossible to be more precise about the intuitive semantics that might be assigned to a signature than to say that a human being imagines particular sets and particular functions defined on those sets to be the signature names. Then the intuitive abstraction is complete.

The best example of an intuitive abstraction is the natural numbers. The set is an infinite one, containing a distinguished element 0, and there is a generating function S (Successor) that produces the other elements, all distinct: S(0), S(S(0)), ... Other less fundamental operations such as addition and multiplication can be defined. These are all so familiar that to name them is to feel that they are known and understood, exactly the character that intuitive abstractions have. It is important to separate the intuition from any formal treatment (here for example the objects defined by the Peano axioms), since the identity of formal and intuitive objects can never be proved, and we want to avoid an infinite regress by grounding our definitions on what a human being has in mind.

1.1. Implementation

For an implementation we can be precise. A programming language is involved, with a well-defined semantics. The language has some built-in entities, and the ability to define abstractions as extensions. The usual situation is that there are a finite number of primitive types, along with a few fixed ways to construct new types from these. The language has a procedure-definition facility, in which parameters and returned values may be any of the built-in or constructed types. A data-abstraction-defining facility allows a record of built-in types, along with a group of procedures, to be encapsulated and called something like a "class." The record constitutes the internal, hidden representation of the abstraction to be defined, while the procedures alone are visible from the outside, and permit manipulation of this data. In order to give examples, we must have a particular syntax, so we construct one in a Pascal-like fashion. For example,

```
class Prize
  record
    Thing:  (old, new, borrowed, blue);
    Value:  int
  end record;
  func Val(T, U: Prize): int;
    begin
      ...
    end;
  func Build(I: int): Prize;
    begin
      ...
    end;
  ...
end class
```

The only aspect of such a program fragment that is not defined by the programming language, independent of its abstraction facility, is the restricted visibility of the unnamed record that begins the class. Within the class this record takes the class name (Prize in the example) and may be manipulated normally. Outside the class the name may only be used to type declarations; the outside program may not refer to the components. Indeed, the using program is not to have any idea of what constitutes the internal record. Objects of the defined type can only be manipulated by passing them as parameters to the procedures within the class. (It is possible to use the defined type without declaring any objects. For the example above, it might make sense to evaluate Val(Build(3), Build(-3)) in which the hidden data storage has only a fleeting existence.)

In most of the existing data-abstraction programming languages the internal record (here unnamed) retains values from call to call of the procedures encapsulated with it. That is, a persisent internal state of the class exists, and each declared object of the new type receives a unique version of this state. The actual situation can be captured formally by treating the internal record as an additional phantom parameter and result, for each procedure. For simplicity we instead define our Pascal-like language so that its class records are local: on each procedure invocation the record comes into existence, and must be initialized to be used sensibly; no values are retained from call to call.

The meaning of a new data abstraction is well-defined by the programming-language semantics: the appropriate records and other built-in quantities are manipulated by the procedures just as if the class boundaries were not present. It remains only to give the obvious correspondence with the syntax of the signature of an abstraction: the record within a class goes with one set name of the signature, and the procedures go with the signature's function names, with the domains and ranges matched up as declared. The built-in types also match with set and function names of the signature. Once a class has been defined in this way, the new type (defined by the class name and corresponding to the unnamed internal record) may be employed in defining other classes just as the built-in types can. The most natural way to construct a complex definition is as a strictly nested hierarchy, beginning with a class that uses only built-in types; however, there is no reason to forbid mutual interactions except those that are not well defined. Situations in which nothing is really defined can be detected in syntax just as nonsensical recursive types are detected [van Wijngaarden et al. 76].

To discuss precisely the meaning of a class the programming-language semantics must take a precise form, and we select that devised by Harlan Mills [Linger, Mills & Witt 79]. Each built-in primitive type corresponds to a set of intuitive objects; for example, int to the integers, enumerated types to finite sets, etc. Record types correspond to cross products of the sets of their components. Procedures correspond to mappings among the sets to which their parameters and result values correspond. Thus for example, in the class Prize above the objects are integers Z , a certain four-element set W for the enumerated type, and for the special record, pairs from Z x W . The functional objects are

```
[Val]:   (Z x W) x (Z x W) ---> Z
[Build]:  Z ---> Z x W .
```

The felicitous notation of surrounding the procedure syntax name by brackets to indicate the meaning function was invented by Kleene [Kleene 52]. The most interesting and difficult part of the semantic definition of the programming language is here omitted, in which it is spelled out which particular functions [Val] and [Build] happen to be. The particular ones are determined by the meaning of the (elided) bodies of the procedures. Roughly, the proper function is that one that agrees with the collection of input-output values arising from the procedure execution. To fill in the details of the definition requires consideration of how each computational feature of the language works. A definition along these lines for a simple language is given in complete detail in [Hamlet 78].

A _class_ of the programming language can now be made to provide a semantics for a signature, by matching up its types with the set names, and its procedures with the function names. If this can be done consistently, the class can be said to _implement_ the signature, and give it a meaning, the implemented abstraction. Any given signature can be implemented in many ways, because procedure bodies in the code are not constrained by the necessary correspondence. Similarly, any _class_ implements some signature, namely the one in which the names are simply taken from its types and procedures. For example, the _class_ Prize above implements a signature with two set names s and t and two function names f and g, such that $f: s \times s \longrightarrow t$ and $g: t \longrightarrow s$.

The _correctness_ of an implementation must be relative to an independent, intuitive idea. If we have an intuitive abstraction in mind, then an implementation abstraction with the same signature is correct if the meaning of the programming-language entities mimics that of the intuitive ones exactly. That is, each function computed by the program must agree with the corresponding intuitive function. Unfortunately, this agreement can be attained only through an additional intermediary, because the function the program computes, and the intuitive functions, have different domains and ranges. In an intuitve abstraction with the signature of the above example, let S be the set which is implemented in the _class_ record, and let Z be the integers (which _int_ implements). Suppose that the function $g: Z \longrightarrow S$ is the one we have in mind corresponding to the procedure Build. (These symbols stand for actual sets and functions now, not for the content-free names of the signature.) Correctness must then mean that [Build] -- the function computed by the procedure -- agrees with g. But [Build] does not have S as its range; rather it has a cross product set that is the meaning of the _class_ record. What is missing is a correspondence between elements of the class record and elements of the intuitive set S. We know that these sets correspond by name, but without an exact description of which element corresponds to which, it is impossible to state the condition that [Build] and g agree. In the example, the range of [Build] is $Z \times W$, so the missing intermediary is a mapping that associates an element of this set with an element of S. Let this mapping be $R: Z \times W \longrightarrow S$ (for Representation -- the programming-language pair represents the abstract object). Correctness then means that [Build] and g agree as best they can: that given any $z \in Z$, the intuitive function and the implementation produce results that agree. In symbols for this case,

$$g(z) = R([Build](z)) .$$

A more graphic way to present the same statement is in an <u>implementation diagram</u> for Build:

$$\begin{tikzcd}
Z \arrow[r, "g"] & S \\
Z \arrow[r, "{[Build]}"] & Z \times W \arrow[u, dashed, "R"']
\end{tikzcd}$$

If the diagram commutes, this part of the implementation is correct.

When the formal definition of correct implementation is presented in Section 4, we will see that the representation must be taken to be technically more complicated, in order to capture the idea of implementing one abstraction in terms of another. In the example, the simplification is that we have taken Z (the integers) to be the same set as a programming-language meaning, and an intuitive abstraction. Where this simple correspondence fails, a cross product of representation functions is required to construct the implementation diagram. Behind the technical details lies the important idea that in building a hierarchy of abstractions, lower levels may be imperfectly understood. Formally, this imperfect understanding enters through selecting a peculiar representation to map the lower-level abstractions within the higher-level implementation diagram.

As the idea of "correctness" is framed above, it describes a relationship among three independent ideas: an intuitive abstraction, a program, and the representation that links their sets. In practice, the intuitive abstraction is the starting point, and with some representation in mind, a program is written. What should not escape notice is that when the three ideas do not match to yield technical correctness, the fault may lie with the code, but it may also (independently) lie with the representation. For a fixed representation there may be any number of correct implementations (including zero); for fixed code there may be any number of correct representations (ditto). In Section 4 we will see that in the practical case where the code is fixed, a collection of interrelated correct implementations results as the representation varies. Because this collection has a simple, tight structure, we propose it as a good way to think about groups of abstractions.

1.2. Specification

By "specifications" for a data abstraction, we mean a mathematical formalism that describes the sets and functions named in a signature. In mathematics itself, the discipline of "foundations" is concerned with giving formal meaning to intuitive objects. Set theory and number theory have been most extensively treated, using the tools of mathematical logic. Abstract algebra is also a foundational technique, but it permits use of definitional phrases such as "the unique object such that..." which cannot always be justified by a construction. For the specifications to be used here, we combine the algebraic approach with logical ideas to give the relationship with intuitive abstractions.

Just as there are many programming languages that allow the implementation of data-abstraction extensions to their types, so there are many candidates for an algebraic specification formalism. These all have their basis in formal theories. These theories have a syntax of <u>well-formed formulas</u> (<u>wfs</u>) that are constructed from given collections of symbols. Wfs are a syntactic idea, in that no meaning is attached to a particular formula, but there are precise rules for a formula's construction.

From the many choices available, we select a particular formalism, essentially that of [Guttag 77], but without conditional axioms. The symbols to be used are:

(1) Several collections of distinct variables, each unlimited. Each collection will be used with a set name from a signature (its <u>type</u>) to construct a finite set of finite wfs. Thus in any

particular case a finite number of variable collections, each containing a finite number of variables, suffices.

(2) A collection of function symbols, whose number is similarly unlimited/limited.

(3) A collection of special 2-ary equality-function names, each having distinct domain $D \times D$ for some set name D, and a common range (named Boolean), written (infix) $=_D$.

(4) Parentheses and commas.

From these symbols the usual "terms" can be constructed. Each term has a "type" that corresponds to a set name. A variable is a term, of the type associated with its collection. A function name is a term, of the type of its range, provided that it is followed by a list of parameters that are terms of the appropriate types to match its domain, enclosed in parentheses and separated by commas.

Certain terms are distinguished as "axioms." These are of type Boolean, having one of the special equality-function names as the outermost symbol. All variables appearing in an axiom are universally quantified by default.

The way in which we use the formalism, and the general method of giving its correspondence with intuition, do not depend on these details, but some of the properties of the formalism would change if they were changed. In any case, a specific form is needed for examples.

For example, the following are axioms,

$$g =_U u$$
$$f(t, g) =_T t$$
$$f(t, u) =_T f(f(t, v), v)$$

if for two set names T and U (along with Boolean B), t is a variable of type T, and u, v are of type U; and, there are two function names $f\colon T \times U \longrightarrow T$, and $g\colon \longrightarrow U$; with special equality functions $=_T\colon T \times T \longrightarrow B$ and $=_U\colon U \times U \longrightarrow B$. In the second axiom, $f(t, g)$ is a term of type T, because f is a function name with this range, t is a variable of type T; and g is a term of type U, because that is its range, and it has no parameters.

The correspondence between a signature, with its set and function names, and a set of axioms, is obvious. Should it happen that there is a consistent correspondence between the appropriate names, we say that the set of axioms is a _specification_ for the signature. This notion is the analogue of "implementation" of a signature without reference to any intuitive meaning; and as in the former case, since the substance of the axioms is not constrained, many axiom sets specify any signature, and any axiom set specifies some signature.

The meaning of a set of axioms, which defines a specified abstraction analogous to the implemented abstraction that a programming language defines, is not so well known in computer science. This meaning is supplied by the concept of a "model" from mathematical logic. By identifying the strictly formal names in axioms with particular set elements and functions, and in particular identifying "Boolean" with the set {_true_, _false_} and the special equality functions with actual equality relations, we can talk about the idea of "truth."

> _DEFINITION_ -- An _interpretation_ of a set of axioms is a mapping that assigns an actual set to each type, and actual functions over appropriate cross products of these sets to the function names. In particular, Boolean must be assigned {_true_, _false_}, and each "=" must be assigned that function which takes the value _true_ if and only if the equality relation for the appropriate set holds. An axiom is _true in an interpretation_ if and only if no matter which set elements under the interpretation are taken for its variable symbols, the functions of the interpretation, applied to these elements, do yield _true_ for the outer "=" function. The sets and functions of an interpretation that makes every axiom in a specification true, is called a _model_ of that specification.

We take this idea as the semantics for specifications: the specified abstraction is any model of the axioms.

For a particular signature, we can now consider another pair of meanings, just as we did for implementation. A set of axioms that define a specified abstraction can be said to be _correct_ relative to another, intuitive abstraction for the signature, if the intuitive abstraction agrees with those axioms. The technical form of this statement is just that the intuitive abstraction is one of the models for the axioms. That is, there must exist an interpretation mapping that carries the axiom formalism onto the intutive sets and functions, such that the axioms are true in that interpretation.

An interpretation is as essential to correctness of a specification as a representation is to implementation. (Indeed, [Hoare 72] uses the word from logic for the latter idea.) As before, it can happen that axioms fail to correctly specify a given intuitive abstraction, not because of an error in the axioms, but because the correspondence of the interpretation is in error. However, there is less leeway for committing interpretation blunders in specifications, because the interpretation domain has no named structure. A representation mapping carries a complex record object to an intuitive one, and the map is likely to depend on the record structure details; the interpretation maps only homogeneous sets.

1.3. Implementation vs. Specification

Data abstractions are implemented or specified by people beginning with an intuitive idea of the objects and operations desired. The definitions of correctness above relate the resulting formal objects to what a person had in mind. But in practice, only the formal objects exist for analysis. We raise the question of the extent to which they overlap:

> Suppose that a certain <u>class</u> implements a number of intuitive abstractions correctly, and a certain set of axioms correctly specifies a number of intuitive abstractions. If they share the same signature, what is the relationship between the classes of intuitive abstractions they correctly capture?

This question implicitly raises the deeper question of the structure of the correct intuitive abstractions for a given implementation or specification.

Because the details of the formalisms for implementation and specification are so different, we cannot compare them without a common theoretical framework. The remainder of this paper is devoted to analysis of such a framework, a word algebra of constants.

2. Word Algebra

The syntax common to an intuitive abstraction, an implemented abstraction, and a specified abstraction is captured in the collection of set and function names called a signature.

> DEFINITION -- A signature is a pair (S, F) consisting of a finite, nonempty collection of set names S (the sets are called sorts) together with a finite, nonempty collection of function names F, the function domains being cross products of sorts, and the ranges sorts. The tuple of types of the parameters of a function is called its arity. Function names are considered to include arity information. The notation "f: D ---> R" indicates that D is the domain of f and R its range. When the arity is important, D will be written out as a cross product to show it. One sort is the distinguished sort-of-interest.

In talking about a signature, it is important to remember that only names are involved. For example, the signature gives not the sorts, but only the sort names. The distinguished name of the sort-of-interest is intuitively the name of the abstraction itself; the objects of this sort are the ones to be defined by the abstraction that uses the names from the signature. Fig. 2.1 contains the signature for the List data abstraction, an example we will use in later sections.

```
SORTS

  List
  Elt

FUNCTIONS

  EmptyL:                     --> List
  Conc:      List x List      --> List
  Head:      List             --> Elt
  Tail:      List             --> List
  Makelist:  Elt              --> List
  One:                        --> Elt
```

Figure 2.1 The signature of the List data abstraction

Given a signature of a data abstraction there exists an algebraic structure of possible meanings of that signature, a lattice of semantic interpretations derived solely from the signature.

DEFINITION -- The constant word algebra W_c of a signature (S, F) is the set of all constants formed as follows:

(1) Each 0-ary function $f: \longrightarrow S_1$ in F is a constant word of type S_1.

(2) If $f: S_1 x \ldots x S_{n-1} \longrightarrow S_n$ is a function in F and $w_1, \ldots, w_{n-1}$ are constant words of types $S_1, \ldots, S_{n-1}$, then $f(w_1, \ldots, w_{n-1})$ is a constant word of type S_n.

(3) Nothing else is a constant word.

For example, given the signature of Fig. 2.1, the following are constant words:

EmptyL

Makelist(One)

Conc(Makelist(One),Makelist(One)) .

An algebra is a pair (S, F), where S is a collection of sets, and F is a collection of mappings between the sets. The sets of W_c are named by the sorts of the signature and the mappings are named by the function names of the signature. The reason for defining W_c is to have a name for each value of each sort. No matter which view of data abstractions is taken (intuition, specification or implementation), one has to have a way of describing the elements of sorts. W_c provides a name for every element that is the result of some sequence of operations. If one's intuitive view of a data abstraction includes elements of sorts that cannot be generated by sequences of operations, then W_c cannot describe those elements. However, programming languages that support type encapsulation do not allow implementations of such data abstractions. (Initialization of variables is an operation in our view.)

2.1. Equality in Wc

Two different sequences of operations may be intended to produce the same value, the same element of a sort. The two corresponding elements of W_c

should be equal. We call such intentions semantic interpretations. In order to define this idea precisely we need some mathematical terminology.

DEFINITION -- An equivalence relation $\sim$ over a set X is a binary relation satisfying the following properties, for all x, y, z in X :

(1) $x \sim x$. (Reflexive)

(2) If $x \sim y$ then $y \sim x$. (Symmetric)

(3) If $x \sim y$ and $y \sim z$ then $x \sim z$. (Transitive)

A congruence relation on an algebra (S, F) is a set $\{\sim_i\}$ of equivalence relations, one relation defined on each set $S_i \in S$, with the substitution property:

(4) For all functions $f: S_1 x \ldots x S_{n-1} \longrightarrow S_n$, $x_i \sim_i y_i$, where $i = 1,\ldots,n-1$, implies $f(x_1,\ldots,x_{n-1}) \sim_n f(y_1,\ldots,y_{n-1})$.

Now that we have the terminology, we say precisely what we mean by a semantic interpretation.

DEFINITION -- A semantic interpretation of a signature is a congruence relation on its constant word algebra W_C . We say that two values of W_C are equal in a semantic interpretation whenever they are related by that congruence relation.

We have deliberately chosen an algebraic definition for "semantic interpretation," because we wish to describe intuitive data abstractions that have been correctly specified by algebraic axioms. Perhaps it is a surprise that this definition applies equally well to implementations, as we will show in section 4.

Each semantic interpretation, because it is a congruence relation, defines a unique algebra, called a quotient algebra. One such quotient algebra behaves just like any intuitive data abstraction for a signature. That is, it has the right number of elements in each sort, and each function produces the right value for each set of input values. We will abuse the terminology slightly, and refer to a congruence relation as a surrogate for a data abstraction that one might have in mind. It would be more precise to say that the quotient algebra defined by the congruence relation is the surrogate. Where there is no chance of confusion we will even drop the "surrogate for" terminology and use the term "semantic interpretation" when we mean an intuitive data abstraction. We summarize these conventions as follows:

ASSUMPTION -- Every intuitive data abstraction with signature S can be modelled by a quotient algebra that is defined by a congruence relation on the constant word algebra W_c of S. That is, every intuitive data abstraction has a surrogate.

2.2. Structure of Semantic Interpretations

Each congruence relation on W_c may be thought of as the set of all pairs of constants that are equal in that relation. We thus order congruence relations by set containment: a congruence relation is contained in another congruence relation if and only if it is a subset of the other. A semantic interpretation is contained in another semantic interpretation if all the constants that are equal in the first are equal in the second. To capture this ordering relationship precisely we need more terminology.

DEFINITION -- A partially ordered set $(A, <)$ is a set A with a relation $<$ satisfying the following properties, for all $a, b, c \in A$:

(1) $a < a$. (Reflexive)

(2) $a < b$ and $b < a$ implies $a = b$. (Antisymmetric)

(3) $a < b$ and $b < c$ implies $a < c$. (Transitive)

A lattice is a partially ordered set in which every two elements have a least upper bound, called the join, and a greatest lower bound, called the meet. A complete lattice L is a lattice in which every subset of L has a join and meet. A complete sublattice is a subset L of a lattice M closed under the join and meet operations defined on M, operating on subsets of L.

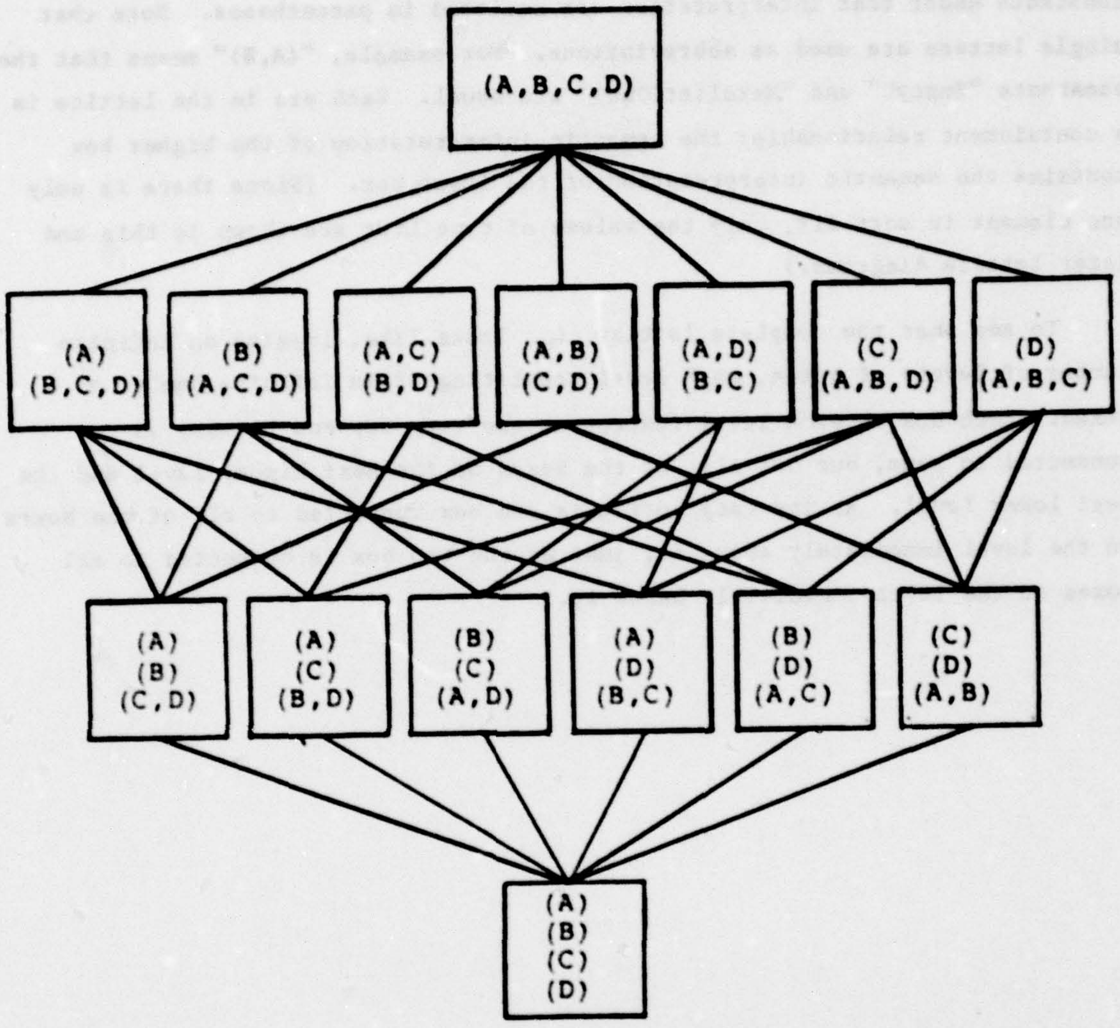
The main result of this section is the following theorem.

<u>THEOREM</u> -- The collection of semantic interpretations of a signature forms a complete lattice, denoted L_W .

<u>PROOF</u> -- By definition the collection of semantic interpretations of a signature is the collection of congruence relations on the word algebra W_C . The collection of congruence relations on an algebra ordered by set containment is known to be a complete lattice [Birkhoff & Lipson 70], with meet and join operations set intersection and congruence closure union. The top element of this lattice is the trivial algebra, containing one element in each sort. $\boxed{\ }$

2.3. Example

As illustration, a part of the lattice of semantic interpretations of the List signature is shown in Fig. 2.2. Even for such a small example, L_W , like W_C , is infinite in size. However, the portion shown is enough for our purposes. Later examples will not need any of the missing pieces. What is shown is a complete sublattice of L_W . Only four constants from W_C and their relationships are shown.



A = EmptyL
B = Makelist(One)
C = Conc(Makelist(One),Makelist(One))
D = Conc(Conc(Makelist(One),Makelist(One)),Makelist(One))

Figure 2.2 A portion of the lattice L_W for List

Each box in Fig. 2.2 represents a semantic interpretation. All equal constants under that interpretation are enclosed in parentheses. Note that single letters are used as abbreviations. For example, "(A,B)" means that the constants "EmptyL" and "Makelist(One)" are equal. Each arc in the lattice is a containment relationship: the semantic interpretation of the higher box contains the semantic interpretation of the lower box. (Since there is only one element in sort Elt, only the values of type List are shown in this and later lattice diagrams.)

To see what the complete lattice L_W looks like, imagine an infinite number of levels of boxes, each level consisting of an infinite number of boxes. Each box on each level (except at the very top and bottom) is connected to some, but not all, of the boxes on the next higher level and the next lower level. At the very bottom is one box connected to all of the boxes on the level immediately above it, just as the top box is connected to all boxes on the level immediately below it.

3. Specification

The formal theory described in Section 1.2 permits us to write collections of axioms using arbitrary variable symbols, function symbols, and special "equality" function symbols. In defining the notion of the "type" of each term, a collection of set names is involved, associated with each collection of variables, and with function domains and ranges. The syntax of the wfs of this formal theory can be made to match the syntax of a signature, setting the stage for giving a signature the meaning of a specified abstraction.

> DEFINITION -- A signature (S, F) is specified by a set of axioms, if and only if each type involved in the axioms corresponds to a member of S in a consistent way (the consistency enters when function domains and ranges are considered), and each function symbol in the axioms corresponds to a member of F .

A signature cannot be specified unless it thus contains a sort name for the Boolean required in the axiom formalism, and all the necessary equality functions for its sort names. (Since the types of the equated terms make clear which equality is involved in any axiom, we henceforth drop the type subscript, and write simply infix "="). With these minimal restrictions, any signature has many specifications, and each set of axioms has a "natural" signature that it specifies, in which the sort and function names are taken from its types and function symbols.

For example, Fig. 3.1 contains a specification whose natural signature has two sort names and five function names as indicated.

```
SORTS

  List
  Elt

FUNCTIONS

  EmptyL:                  --> List
  Conc:      List x List   --> List
  Head:      List          --> Elt
  Tail:      List          --> List
  Makelist:  Elt           --> List
  One:                     --> Elt

AXIOMS

  Head (EmptyL) = One
  Head (Makelist (X)) = X
  Head (Conc (Makelist (X), Makelist (Y))) = X

  Tail (EmptyL) = EmptyL
  Tail (Makelist (X)) = EmptyL
  Tail (Conc (Makelist (X), Makelist (Y))) = Makelist (Y)

  Makelist (ErrorN) = EmptyL

  Conc (EmptyL, L) = L
  Conc (Makelist (X), EmptyL) = Makelist (X)
  Conc (Makelist (X), Conc (Makelist (Y), Makelist (Z))) =
                      Conc (Makelist (X), Makelist (Y))
  Conc (Conc (Makelist (X), Makelist (Y)), L) =
     Conc (Makelist (X), Makelist (Y))
```

Figure 3.1 Specification of List data abstraction

3.1. Correctness

As described in Section 1.2, we take the meaning of a specification to be any model of the axioms, making it possible to define agreement with intuition:

DEFINITION -- Given an intuitive data abstraction A and a set of axioms that specify the same signature, we say that the axioms correctly specify A if and only if A is a model for them. That is, if and only if there exists an interpretation of the axiom formalism into A that makes them all true.

Because an intuitive abstraction and a specified abstraction are required to share a signature, the only way that the former can fail to be a model is by the assignment chosen for the function symbols: if the intuitive functions do not in fact satisfy the axioms, the specification is incorrect.

A specification contains semantic information in the form of axioms. The axioms may be viewed as a list of requirements that must be satisfied by any semantic interpretation of the signature. It is our view that these requirements are minimal, but not maximal conditions. A specification does not define a unique semantic interpretation, but a collection of semantic interpretations.

3.2. Semantic Interpretation

In order to describe what a specification means, we need to characterize all the semantic interpretations that agree with the specification. Some interpretations must be ruled out, because they do not equate constants that the specifications asserts are equal. The minimal set of equal constants that must be in a semantic interpretation that ageees with a specification are in the relation speceq.

DEFINITION -- A derivation from a specification S is a finite sequence of equations formed as follows:

(1) $w = w$, where w is any constant of W_c , is an equation.

(2) If $w_1 = w_2$ is an equation then $w_2 = w_1$ is an equation.

(3) If $w_1 = w_2$ and $w_2 = w_3$ are equations, then $w_1 = w_3$ is an equation.

(4) An equation is formed from an axiom of S by an assignment of constants to variables, where each occurrence of a variable x of type S in A is consistently replaced by a constant w of type S .

(5) If $w_1 = w_2$ and $f(\ldots,c,\ldots) = f(\ldots,c,\ldots)$ are equations, and c is of the same type as w_1 and w_2 , then $f(\ldots,w_1,\ldots) = f(\ldots,w_2,\ldots)$ is an equation.

(6) Nothing else is an equation.

The last equation in a derivation is the equation <u>derived</u>. Two elements w_1 and w_2 of the constant word algebra W_c of a specification S are in <u>speceq</u> if and only if the equation $w_1 = w_2$ can be derived from S . We say that w_1 and w_2 are <u>equal</u> in speceq .

Our intuitive view of specification of a data abstraction is: two values of a sort are equal because they are identical, because an axiom asserts they are equal, or because they were created by substitution of equal values for the same part of equal values. The rules for forming equations in a derivation capture just those equalities: rule (1) for identical values, rule (4) for axioms and rules (2), (3) and (5) for substitutions.

<u>LEMMA</u> -- Speceq is a congruence relation on W_c .

<u>PROOF</u> -- Rules (1), (2) and (3) for forming equations in a derivation are just restatements of the reflexive, symmetric and transitive properties of an equivalence relation. Thus speceq is an equivalence relation. To show that it is a congruence relation we must demonstrate that the substitution property holds. Suppose constants w_1 and w_2 of type S_i are equal in speceq . Then there exists a derivation of $w_1 = w_2$. Let $f: S_1 x \ldots x S_i x \ldots S_{n-1} \dashrightarrow S_n$ be a function. The constant word

algebra W_c contains all values of f, and all the elements of W_c are constants. So, there exist constants $c_1,\ldots,c_{n-1}$ of types $S_1,\ldots,S_{n-1}$ and a constant

$$f(c_1,\ldots,c_i,\ldots,c_{n-1}) .$$

Continuing from the derivation of $w_1 = w_2$, by rule (1) we derive the equation

$$f(c_1,\ldots,c_i,\ldots,c_{n-1}) = f(c_1,\ldots,c_i,\ldots,c_{n-1}) .$$

Finally, we derive

$$f(c_1,\ldots,w_1,\ldots,c_{n-1}) = f(c_1,\ldots,w_2,\ldots,c_{n-1})$$

by rule (5). $\boxed{}$

So, speceq is a semantic interpretation. Next, we show that it is correctly specified.

<u>LEMMA</u> -- Given a specification S, its semantic interpretation speceq is a model for S.

<u>PROOF</u> -- This is one place where we cannot afford the confusion that might arise from treating a congruence relation, speceq, like an algebra. Let us call the quotient algebra defined by speceq <u>specalg</u>. The sets of specalg are sets of equivalence classes of elements of W_c. That is, all elements of W_c that are equal in speceq are in one equivalence class in specalg. The mappings of specalg are functions defined by relations in speceq : Let $f: S_1 x \ldots x S_{n-1} \longrightarrow S_n$ be a function in the signature of S. Let $S_i = \{w_{i_j}\}$ for $i = 1,\ldots,n$. The set of all pairs $(f(w_{1_j},\ldots,w_{n-1_j})\ w_{n_j})$ in speceq defines a function f' in specalg.

We now show that specalg is a model of S. Let $w_1 = w_2$ be any instance of any axiom of S formed by consistently replacing all variables by constants of W_c. By rule (4) for forming equations, $w_1 = w_2$ can be derived. That means that the functions of specalg whose names appear in v_1 and v_2 operate in such a way that "$v_1 = v_2$" has tha value "true" in the Boolean sort of specalg. So, specalg is a model of S. $\boxed{}$

3.3. Structure of Semantic Interpretations

The congruence relation speceq is just one semantic interpretation of a signature. There may be others that agree with a specification.

DEFINITION -- A congruence relation is said to satisfy a specification if and only if it contains (in the set-theoretic sense) the congruence relation speceq for that specification.

We can now characterize correct semantic interpretations.

THEOREM -- Given an intuitive data abstraction A, its surrogate semantic interpretation A′ and a specification S with the same signature, A is correctly specified by S if and only if A′ satisfies S.

PROOF -- First, we show that satisfying S implies correctness. Let $w_1 = w_2$ be any instance of any axiom of S formed by consistently replacing all variables by constants of W_c. The equation "$w_1 = w_2$" can be derived by rule (4) of the definition of derivations. So, w_1 and w_2 are equal in speceq. A′ contains speceq, so w_1 and w_2 are equal in A′. Thus, every axiom in S is true in the interpretation A′. Therefore, A is correctly specified by S.

Now we show that correctness implies satisfaction of S. Suppose the contrary: A is correct, but A′ does not contain speceq. Then, there exists an equality $w_1 = w_2$ in speceq that is not in the relation A′. Let $E_1,\dots,E_n$ be the sequence of equations in a derivation of $w_1 = w_2$. Suppose an equation E_i : $v_i = v_i$ is constructed by rule (1). Then the equality "$v_i = v_i$" is true in A′, because A′ is a congruence relation and admits all identities of that form. Similarly, if E_i is constructed by any of the rules (2), (3) or (5), then it is a statement of equality that must be true in the congruence relation A′. Suppose E_i is constructed by rule (4). Then, E_i is formed by substituting constants for variables in an axiom of S. But, every axiom is true in A′, because A is correctly specified. In every case, E_i is a statement of equality that must be true in A'. Therefore, the last equation, "$w_1 = w_2$", must be true in A'. $\boxed{\ }$

When is an intuitive data abstraction correctly specified? If the semantic interpretation satisfies the specification, then all of the values that the specification asserts are equal must be equal in the intuitive abstraction. However, there may be values that are equal intuitively, but are not necessarily equal according to the specification. Even though the specification does not completely describe the abstraction, nothing in the specification is contradicted. In this case we say that the specification is still correct.

Our acceptance of "incomplete" specifications as correct is motivated by the <u>use</u> of data abstractions. In practice, the use of an abstraction will almost never require every distinct value of each sort. Those values that are never used might just as well be identified with one value in each sort that is used.

A particularly *perverse* case of correct incomplete specification is the use of a data abstraction that does not distinguish between any values in any sorts. For example, an abstraction that includes input/output operations may be used in a program only to read and write values of that abstraction. Such use is characterized by the trivial interpretation that equates all values of each sort. Even the values "true" and "false" of the sort Boolean are equated in this interpretation. Although the distinctions that the specification makes between values of this abstraction are not made by the use of the abstraction, it would not be right to say that the specification is incorrect.

With the help of the previous theorem we obtain a characterization of all correctly specified data abstractions of a specification.

<u>THEOREM</u> -- The collection of data abstractions that are correctly specified by a specification form a complete sublattice of L_W. We denote the sublattice L_s.

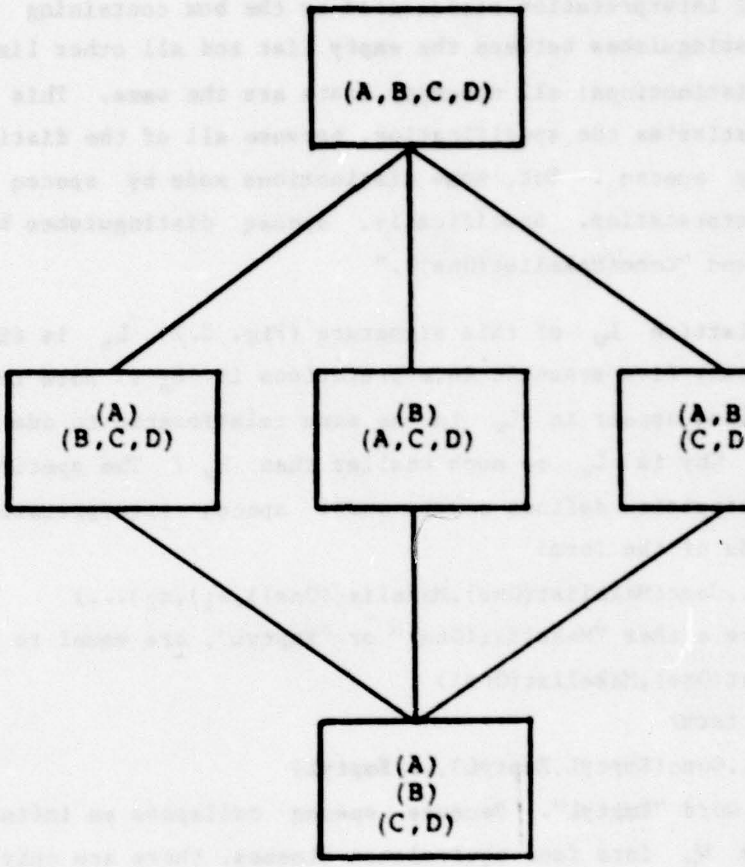
<u>PROOF</u> -- By the previous theorem we may argue about correctly specified abstractions in terms of the surrogate semantic interpretations that satisfy the specification. Each semantic interpretation is a congruence relation on W_c. The congruence

relations that satisfy the specification contain the congruence relation speceq , by definition of satisfy . The elements of L_s are contained in L_W , because they are all congruence relations on W_C . Therefore, the lattice operations of L_W apply for L_s .

We need to show that the meet and join of every subset of L_s is contained in L_s . The intersection of any number of sets containing a common subset contains that subset. So, the meet of any number of congruence relations containing speceq is a congruence relation containing speceq . Similarly, the congruence closure union of any number of congruence relations containing speceq is a congruence relation containing speceq . $\square$

3.4. Example

Fig. 3.2 depicts the lattice L_s of the List data abstraction specified in Fig. 3.1. The semantic interpretation speceq is represented by the box at the bottom of the lattice. Under this interpretation there are three distinct values of type List: "EmptyL," "Makelist(One)," and the value "Conc(Makelist(One),Makelist(One))," which is equal to "Conc(Conc(Makelist(One),Makelist(One)),Makelist(One))." These last two values are equal because of axiom L11. (Recall our convention not to show the single value "One" of sort Elt in any of the boxes.)



A = EmptyL
B = Makelist(One)
C = Conc(Makelist(One),Makelist(One))
D = Conc(Conc(Makelist(One),Makelist(One)),Makelist(One))

Figure 3.2 The lattice L_s for List

The semantic interpretation represented by the box containing "(A),(B,C,D)" distinguishes between the empty list and all other lists, but makes no other distinctions: all nonempty lists are the same. This interpretation satisfies the specification, because all of the distinctions it makes are made by speceq . But, some distinctions made by speceq are not made by this interpretation. Specifically, speceq distinguishes between "Makelist(One)" and "Conc(Makelist(One))."

Unlike the lattice L_w of this signature (Fig. 2.2) L_s is finite. In fact, there are only five semantic interpretations in L_s . Note that these five interpretations appear in L_w in the same relationship to one another as they do in L_s . Why is L_s so much smaller than L_w ? The specification of the List data abstraction defines a very small speceq interpretation. For example, all words of the form:

Conc(Conc(...Conc(Makelist(One),Makelist(One)),x_1),x_2)...)

where the x_i are either "Makelist(One)" or "EmptyL", are equal to the word:

Conc(Makelist(One),Makelist(One)) .

All words of the form:

Conc(Conc(...Conc(EmptyL,EmptyL),...EmptyL)

are equal to the word "EmptyL". Because speceq collapses an infinite number of constants from W_c into four equivalence classes, there are only four other interpretations that can contain speceq .

All the semantic interpretations of a data abstraction that satisfy a specification are correctly specified. However, all but speceq are <u>over-specified</u>. That is, given one of these over-specified interpretations, there exists a specification S′ with semantic interpretation speceq′ equal to that interpretation. The lattice L_s' of S′ is smaller than the original lattice L_s . In the example, the semantic interpretation represented by the box containing "(A),(B,C,D)" satisfies a specification S′ whose lattice L_s' contains just the two interpretations labelled "(A),(B,C,D)" and "(A,B,C,D)." The semantic interpretation represented by the box containing "(A),(B),(C,D)" is <u>under-specified</u> by S′ , because it is not in the lattice L_s' . Under-specified data abstractions are incorrectly specified. To reiterate, a data abstraction is over-specified by a specification if it is in the lattice L_s of the specification but is not the semantic interpretation speceq of that specification. A data abstraction is under-specified if it is not contained in the lattice L_s of that specification.

3.5. Relationship to Other Work

The idea of using a lattice to capture the structure of semantic interpretations of algebraic specification of data abstractions is not new. [Giarratana et al. 76] and [Polajnar 78] describe similar lattices, but with much more attention to mathematical details. A difference between L_s and their lattices is the presence of different interpretations of sorts other than the sort-of-interest in L_s. Consequently, L_s is much bigger, containing such degenerate interpretations as the trivial interpretation, which equates all values to one another in each sort.

4. Implementation

The Pascal-like programming language indicated in Section 1.1 allows us to write class definitions containing certain typed objects (in the programming-language sense), and with procedures using these types as parameters and results. The language syntax can therefore be matched to the syntax of a signature in the obvious way.

DEFINITION -- A class definition involving a collection of types T , and procedures P typed from T and using T parameters, is said to implement a signature (S, F) if and only if a correspondence can be established between the names in S and F and the intuitive objects that are the meanings of the types and procedures in T and P . The correspondence must be consistent in assigning procedures with typed parameters to function names with corresponding sort domains, etc. The sort-of-interest must correspond to the type with the class name.

As defined, a given class implements a natural signature whose names are taken from the programming-language syntax itself. Any given signature has many implementations, corresponding to the arbitrary content of the programming-language procedures, and to the arbitrary content of the record that represents the sort-of-interest. For example, the class in Fig. 4.1 implements the same signature specified by the axioms of Fig. 3.1.

```
class List

record
  Values: array [1..Listsize] of Elt;
  Liststart: int;
  Listlength: int;
end record

func EmptyL: List;
  var
    Result: List;
  begin
    Result.Listlength := 1;
    Result.Liststart := 0;
    Result.Values [0] := 1;
    EmptyL := Result;
  end;

func Conc (First, Second: List): List;
  var
    Result: List;
    Pos: int;
  begin
    if ListEqual (First, EmptyL)
      then Conc := Second
      else if ListEqual (Second, EmptyL)
        then Conc := First;
    Result.Liststart := 0;
    Result.Listlength := 0;
    Pos := First.Liststart;
    while Result.Listlength < First.Listlength do
      begin
        Result.Values [Result.Listlength] := First.Values [Pos];
        Pos := Next (Pos);
        Result.Listlength := Result.Listlength + 1;
      end;
    Pos := Second.Liststart;
    while Result.Listlength < (First.Listlength +
                                   Second.Listlength)  and
          Result.Listlength < 3 do
      begin
        Result.Values [Result.Listlength] := Second.Values [Pos];
        Pos := Next (Pos);
        Result.Listlength := Result.Listlength + 1;
      end;
    Conc := Result;
  end;

func Head (L: List): Elt;
  begin
    if L.Listlength = 0
      then Head := One
      else Head := L.Values [L.Liststart];
  end;
```

Figure 4.1 Implementation of List data abstraction

```
func Tail (L: List): List;
  var
    Result: List;
    Pos: int;
  begin
    Result := EmptyL;
    Pos := Next (L.Liststart);
    while Result.Listlength < (L.Listlength - 1) do
      begin
        Result := Conc (Result,
                        Makelist (L.Values [Pos]));
        Pos := Next (Pos);
      end;
    Tail := Result;
  end;
func Makelist (N: Elt): List;
  var
    Result: List;
  begin
    Result := EmptyL;
    Makelist := Result;
  end;

func Next (Pos: int): int;
  begin
    if Pos = Listsize - 1
      then Next := 0
      else Next := Pos + 1;
  end;
func ListEqual (First, Second: List): bool;
  begin
    if First.Listlength = 1 or
         Second.Listlength = 1
      then if First.Listlength =
                  Second.Listlength
        then ListEqual := true
        else ListEqual := false
      else if Head (First) = Head (Second) and
                  ListEqual (Tail (First)
                             Tail (Second))
        then ListEqual := true
        else ListEqual := false;
  end;
func One: Elt;
  begin
    One := 1;
  end;

end class
```

Figure 4.1 (cont.) Implementation of List

In the syntactic correspondence between a signature and class code, part of the information of the program goes unused: the details of the record type corresponding to the sort-of-interest. The class name is made to correspond to this distinghished sort, but the record components may be anything. Thus to assign a meaning arising from the code, the correspondence between the cross-product language-meaning of this record and the sort-of-interest must be given. This correspondence is of course arbitrary, since only its domain (the cross-product record set) appears in the program. Thus although the program meaning is unique, defined by the semantics of the programming language as extended in Section 1.1, the abstraction defined is not unique, since each different representation mapping with the proper domain yields a new meaning.

4.1. Correctness

The simplified discussion in Section 1.1 presumes that when a base type of the programming language appears in the signature, the intuitive set of this sort, and the meaning set from the language, are identical. In that case nothing like a representation mapping (other than the identity) is needed for base types to define implementation diagrams and correctness. This simple view does not correspond very well with reality, and does not extend to hierarchical definition of abstractions. When we use (say) int in implementing some abstraction, it is seldom true that we are really using the full set of integers -- more likely some special subset is really involved. (For example, when the int quantities are serving as array subscripts, the actual meaning set is probably limited to the legal bounds.) And, when one previously defined type is employed in the definition of another, which of the many possible representations of the earlier type record is intended? Nothing in the syntax can say. For these reasons we are forced to define correctness of an implementation in terms of a collection of representation functions, carrying each one of the meaning sets from the language onto an intuitive set. At each level, only the sort-of-interest appears as an explicit tuple of values; the other sets have their representations, however.

DEFINITION -- Given a class definition and an intuitive data abstraction whose signature the class implements, and representation functions mapping the meanings of the class types to the sorts, each procedure of the class has an implementation diagram, as follows:

Let p be a procedure of the implementation corresponding to function name $f\colon A_1 \times \ldots \times A_k \longrightarrow A_l$ of the signature, A_l the sort-of-interest. (Other cases are similar, *mutatis mutandis*.) Let the meaning of the *class* record be the set $D_1 \times \ldots \times D_n$, and let C_i be the meaning set of the implementation corresponding to A_i, for $i \neq l$. Let $R_i\colon C_i \longrightarrow A_i$, except that $R_l\colon D_1 \times \ldots \times D_n \longrightarrow A_l$. The implementation diagram for p is:

$$\begin{tikzcd}
A_1 \times & \ldots \times A_k \arrow[r, "f"] & A_l \\
D_1 \times \ldots \times D_n \times \arrow[u, "R_1"] & \ldots \times C_k \arrow[u, "R_k"] \arrow[r, "{[p]}"] & D_1 \times \ldots \times D_n \arrow[u, "R_l"]
\end{tikzcd}$$

The upper part of the diagram describes the intuitive abstraction, while the lower part describes the meaning of the implementation. We say that the implementation is *correct* if and only if all of the implementation diagrams commute.

An implementation abstraction can fail to be correct for an intuitive abstraction only through the semantics of functions and representations. Syntactically there can be no failure to correspond because the abstractions share the same signature. A semantic error in a function is straightforward: the intuitive function is simply not [p], for the procedure p to which it corresponds. However, it is easy to lose sight of the fact that this statement relies on a representation function to bring the intuitive and implemented domain/range into line. The correctness of [p] never appears in isolation, only in composition with the necessary representation conversions. There is, however, a case in which [p] seems to stand on its own: when the representation functions are all identities. If every intuitive and implemented object is the same, then the only errors can be in the code written to manipulate these objects. This special case probably looms larger in our minds when we think about implementations than it ought to: one of the virtues of implementation is that non-identity representations are convenient, and safely hidden in the encapsulated type definition.

4.2. Semantic Interpretation

Just as a specification defines a class of semantic interpretations that satisfy the specification, an implementation defines a class of semantic interpretations that satisfy the implementation. In fact, most of the statements we made about specifications in the previous section can be made about implementations. This is not hard to explain. Specifications and implementations are both descriptions of the same object, an intuitively-understood data abstraction.

The first step in describing semantic interpretations that satisfy an implementation is to define the concept of a derivation for implementations. Rule (4) for forming equations in a derivation (section 3.2) uses an axiom to generate an equation. There are no axioms in an implementation, so this rule can not be used. However, the code for functions that appear in a constant can be executed, and the result can be checked to see if it is the same as other execution results.

> <u>DEFINITION</u> -- The <u>eval</u> function is a mapping from W_c to the intutitive programming-language meaning sets, using the meaning of the functions appearing in the constant:
>
> $$eval(f(w_1,\ldots,w_n)) = [f]([w_1],\ldots,[w_n]) .$$

We can now construct a new rule for derivations:

(4′) If $eval(w_1) = eval(w_2)$ then $w_1 = w_2$ is an equation.

This gives us the following definition:

> <u>DEFINITION</u> -- A <u>derivation</u>′ is a derivation (section 3.2) with rule (4′) substituted for rule (4). The last equation is said to be <u>derived′</u>. Two constants w_1 and $w_2 \in W_c$ are equal in <u>conceq</u> if and only if the equation $w_1 = w_2$ can be derived′.

Conceq plays the role for implementations that speceq plays for specifications. Any semantic interpretation that satisfies an implementation must equate: identical values, values that are equal because the implementing code of both values has the same meaning, and values that are equal because of substitution of equal values for the same part of equal values. To be a valid semantic interpretation it must be a congruence relation on W_c .

LEMMA -- Conceq is a congruence relation on W_c .

PROOF -- The proof that speceq is a congruence relation (section 3.2) does not use rule (4) of derivation. The other rules for forming equations in a derivation′ are the same as for a derivation. Thus, the proof that conceq is a congruence relation is identical to the proof that speceq is a congruence relation. $\boxed{}$

The semantic interpretation conceq is correctly implemented.

LEMMA -- Given an implementation I , its semantic interpretation conceq is correctly implemented by I .

PROOF -- First we show that conceq defines a mapping R_I that serves as a representation mapping of I onto concalg , the quotient algebra defined by conceq . Let $(T_1,\ldots,T_n)$ be a tuple of concrete types that represent the type S . Define R_I on the restricted set of tuples that represent constants in W_c by:

$$\mathrm{eval}(w) = (c_1,\ldots,c_n) \quad \text{implies}$$
$$R_I : (c_1,\ldots,c_n) \longmapsto |w|$$

for some constant $w \in W_c$ of type S and constants $c_1,\ldots,c_n$ of the types $T_1,\ldots,T_n$. The notation "$|w|$" stands for the equivalence class of w in concalg . We also require:

$$R_I(x) = R_I(y) \quad \text{if and only if}$$
$$\mathrm{eval}(w_1) = \mathrm{eval}(w_2)$$

for some words w_1 and w_2 . R_I is well-defined, because $|w_1| = |w_2|$ in concalg whenever $\mathrm{eval}(w_1) = \mathrm{eval}(w_2)$ in I . For tuples $(d_1,\ldots,d_n)$ that do not represent any constant in W_c , the value of R_I may be undefined. These tuples will never occur in practice, and are not significant to the correctness of R_I .

Next, we show that I correctly implements concalg under the mapping R_I . There exists an implementation diagram:

$$\begin{tikzcd}
S \arrow[r, "f"] & S \\
T_1 \times \ldots \times T_n \arrow[u, "R_I"] \arrow[r, "{[f]}"] & T_1 \times \ldots \times T_n \arrow[u, "R_I"']
\end{tikzcd}$$

for each function f in I. Let w be a constant of type S in W_c, with representation $(c_1,\ldots,c_n)$ in I. By definition of R_I,

$eval(f(w)) = (c_1',\ldots,c_n')$ implies

$R_I : (c_1',\ldots,c_n') \mapsto |f(w)|$.

So, the diagram commutes. $\square$

4.3. Structure of Semantic Interpretations

Since the semantic interpretation conceq was defined similarly to speceq, it is no surprise that the relationship of an interpretation satisfying an implementation is the same as the relationship of an interpretation satisfying a specification.

<u>DEFINITION</u> -- A semantic interpretation is said to <u>satisfy</u> an implementation if it contains (in the set-theoretic sense) the congruence relation conceq of that implementation.

We can now describe correctness in algebraic terms.

<u>THEOREM</u> -- Given an intuitive data abstraction A, its surrogate semantic interpretation A' and an implementation I with the same signature; A is correctly implemented by I if and only if A' satisfies conceq of I.

<u>PROOF</u> -- First, we show that satisfying conceq implies correctness. By the previous lemma R_I is a representation mapping that ensures that conceq is correctly implemented by I. Define any representation mapping R_A to be the same as R_I, except that it maps onto equivalence classes of A' instead of conceq. Every equivalence class of conceq is contained in an equivalence class of A'. So, every diagram that commutes for R_I commutes for

R_A .

Next, we show that correctness implies satisfaction of I . There exists a representation mapping R' for which every implementation diagram commutes. Let $w_1 = w_2$ be an equality in conceq . Let $(c_1,\ldots,c_n)$ represent w_1 and $(d_1,\ldots,d_n)$ represent w_2 . Since conceq is correct, $|w_1| = |w_2|$. This means that $\text{eval}(w_1) = \text{eval}(w_2)$. Correctness of A under R' implies that

$$R'(c_1,\ldots,c_n) = R'(d_1,\ldots,d_n) .$$

So, $w_1 = w_2$ in A' . Therefore, every equality in conceq is in A' . $\square$

When is an intuitive data abstraction correctly implemented? Just as with specifications, one knows that an implementation correctly implements a collection of data abstractions. If the intuitive abstraction has a surrogate semantic interpretation that satisfies the implementation, then the intuitive abstraction is correctly implemented.

The user of a data abstraction may not require that the implementation make as many distinctions between values as it does. In such cases the implementation is still correct as long as the appropriate representation mapping is used. That is, the results computed by the implementation must be interpreted by a representation mapping that maps onto the surrogate semantic interpretation of the intuitive data abstraction.

For example, a data abstraction that requires five distinct values of a sort might be correctly implemented by an implementation that can produce ten distinct values of that sort. However, the ten values must be interpreted in such a way that every operation on the interpreted values behaves as the five values would behave. The commutativity of the implementation diagram captures this idea precisely. Note that there may not be any interpretation of the ten values that behaves correctly. The existence of such an interpretation is only guaranteed when the intuitive abstraction satisfies the semantic interpretation conceq of the implementation.

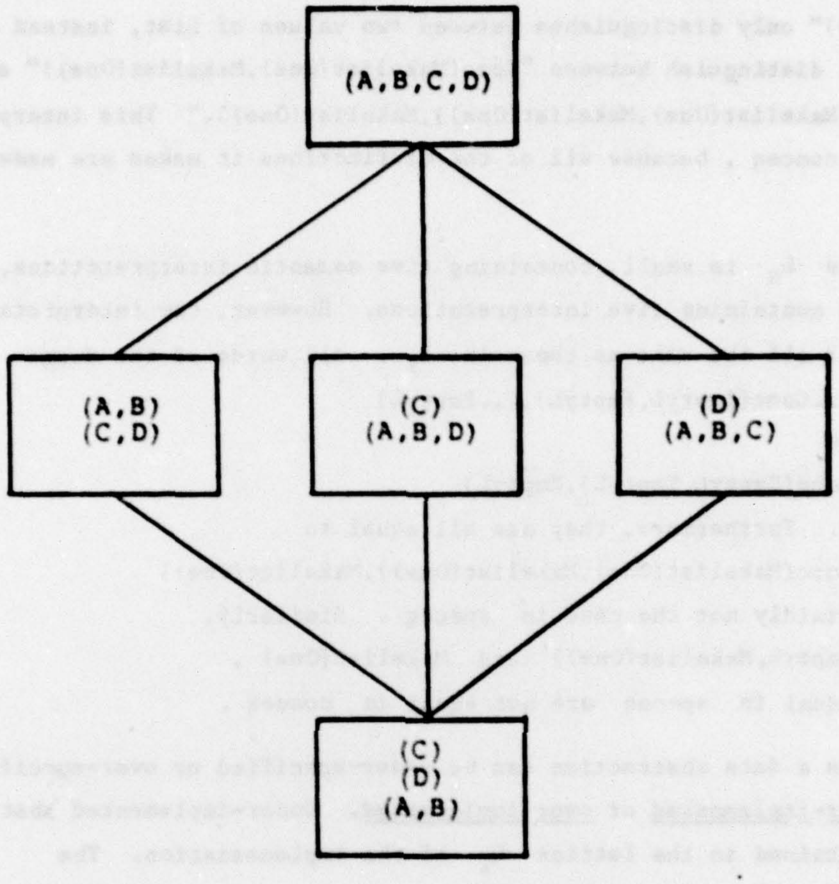
As was the case with specification, we obtain a characterization of all correctly implemented data abstractions.

<u>THEOREM</u> — The collection of data abstractions that are correctly implemented by an implementation form a complete sublattice of L_w . We denote the sublattice L_i .

<u>PROOF</u> -- This proof is identical to the proof that L_s is a complete sublattice of L_w (section 3.3), except that the congruence relations in L_i all contain conceq instead of speceq . $\square$

4.4. Example

Fig. 4.2 depicts the lattice L_i of the List data abstraction implementation in Fig. 4.1. The semantic interpretation conceq is represented by the box at the bottom of the lattice. Under this interpretation there are three distinct values of type List: "Conc(Makelist(One),Makelist(One))," "Conc(Conc(Makelist(One),Makelist(One)),Makelist(One))," and "EmptyL," which is equal to "Makelist(One)." (The code for "EmptyL" does not create an empty list, as its name implies, but a list with one element in it.)



A = EmptyL
B = Makelist(One)
C = Conc(Makelist(One),Makelist(One))
D = Conc(Conc(Makelist(One),Makelist(One)),Makelist(One))

Figure 4.2 The lattice L_1 for List

The semantic interpretation represented by the box containing "(A,B),(C,D)" only distinguishes between two values of List, instead of three. It does not distinguish between "Conc(Makelist(One),Makelist(One))" and "Conc(Conc(Makelist(One),Makelist(One)),Makelist(One))." This interpretation satisfies conceq , because all of the distinctions it makes are made by conceq .

Just as L_s is small, containing five semantic interpretations, L_i is small, also containing five interpretations. However, the interpretations in L_i are not all the same as those in L_s . All words of the form:

Conc(...Conc(EmptyL,EmptyL),...EmptyL)

are equal to

Conc(Conc(EmptyL,EmptyL),EmptyL)

in conceq . Furthermore, they are all equal to

Conc(Conc(Makelist(One),Makelist(One)),Makelist(One)) .

This is certainly not the case in speceq . Similarly,

Conc(EmptyL,Makelist(One)) and Makelist(One) ,

which are equal in speceq are not equal in conceq .

Just as a data abstraction can be under-specified or over-specified, it can be <u>under-implemented</u> or <u>over-implemented</u>. Under-implemented abstractions are not contained in the lattice L_i of the implementation. The implementation is incorrect, because it fails to distinguish between values that can be distinguished in the intuitive abstraction. An over-implemented data abstraction is correctly implemented, but it is not equal to the interpretation conceq of that implementation. There exists an implementation whose interpretation conceq′ is equal to the over-implemented abstraction and whose lattice L_i' is smaller than the original lattice L_i .

5. Intersection of Implementation and Specification

Given a specification and an intuitive data abstraction one can determine whether the intuitive abstraction is correctly specified by the specification: it is if the intuitive abstraction satisfies the specification; otherwise, it is not. In the same way one can determine whether an implementation correctly implements an intuitive data abstraction. Both of these determinations may be difficult to make, because the intuitive abstraction one has in mind is not written down. Comparing the specification to the implementation should be easier. Each can be read and analyzed. In the process of comparison one might see if the intuitive abstraction satisfies either or both.

Our view is that specifications and implementations describe collections of data abstractions. These collections are described by lattices, L_s for specifications, L_i for implementations. We describe the relationship between a specification and an implementation by the overlap of their lattices.

THEOREM -- Given a specification S, its lattice of correctly specified data abstractions L_s, an implementation I, and its lattice of correctly implemented data abstractions L_i; the collection of data abstractions that are correctly specified by S and correctly implemented by I form a complete sublattice of L_s and L_i. We denote the common sublattice SL.

PROOF -- Viewing the lattices L_s and L_i as sets, the intersection of L_s and L_i is a set of semantic interpretations that satisfy both the specification and the implementation. Let SL contain just those interpretations. SL is not empty, because the congruence relation that equates all elements of each sort contains every congruence relation. This trivial interpretation satisfies every specification and every implementation with the same signature.

The lattice operations, meet and join, are the same for L_s and L_i. These operations apply to every subset of SL. Every congruence relation in SL contains the speceq and conceq relations defined by the specification and the implementation. So, the meet and join of any subset of SL are in SL . $\sqcap$

Given a specification and an implementation of a data abstraction, the existence of SL guarantees that there is at least one semantic interpretation that satisfies both specification and implementation. If the intuitive data abstraction one has in mind is in SL then the specification and implementation are correct: they describe the desired abstraction. If the intuitive data abstraction one has in mind is not in SL , then either the specification, the implementation or both are incorrect.

5.1. Case Analysis of SL

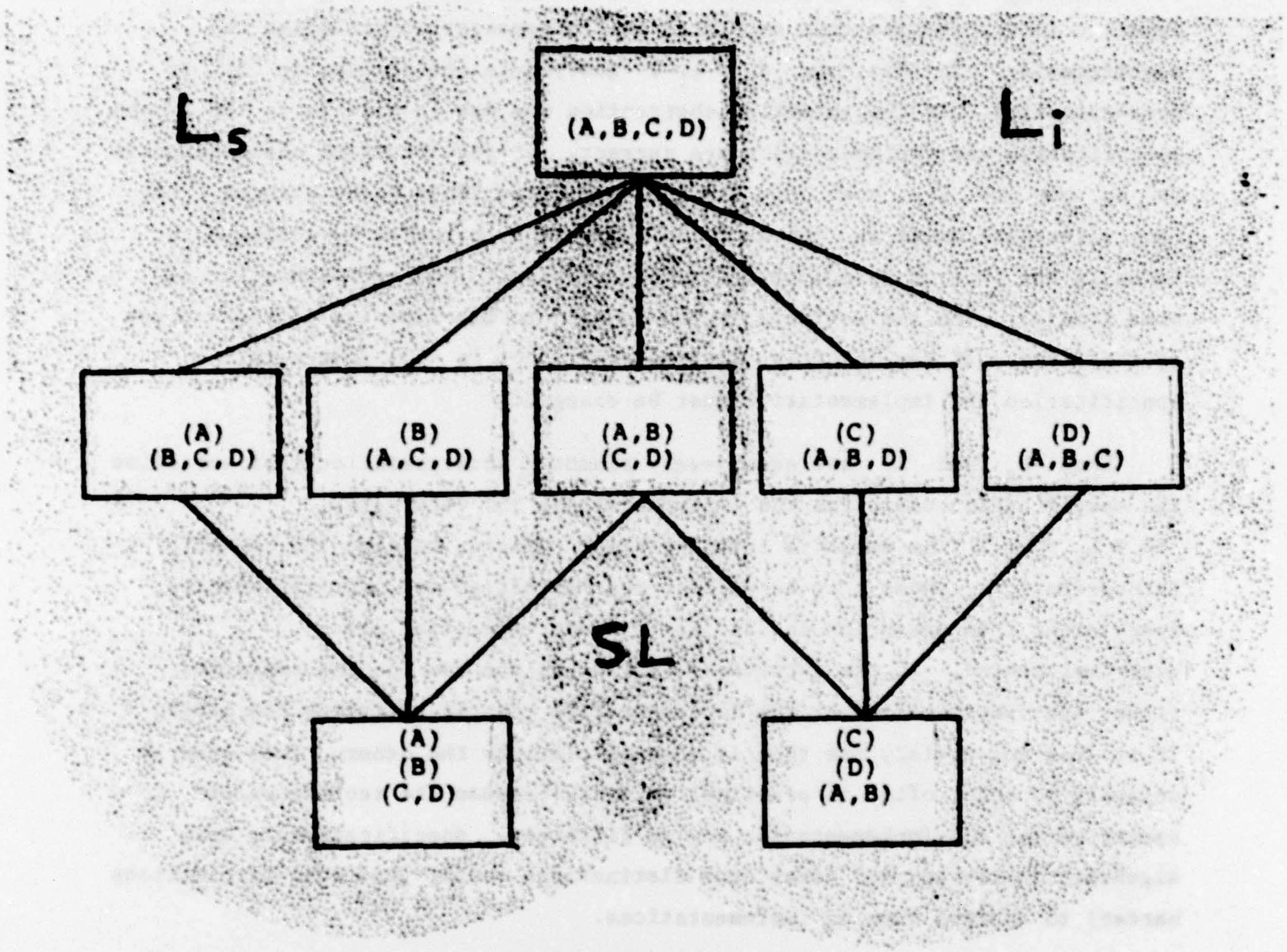
The size of SL relative to the sizes of L_s and L_i sheds some light on the relationship between the specification and the implementation of a data abstraction. There are four possibilities: L_s might be completely contained in, but not equal to L_i ; L_i might be completely contained in, but not equal to L_s ; L_s and L_i might be equal; or L_s and L_i might not be related by containment or equality.

When L_s is completely contained in, but not equal to L_i , every semantic interpretation that satisfies the specification satisfies the implementation. The sublattice SL is equal to L_s . If the intuitive data abstraction one has in mind satisfies the specification it must satisfy the implementation. Every data abstraction that satisfies the specification is over-implemented, because there is some implementation whose lattice L_i' is smaller than L_i and contains the desired abstraction. There are some data abstractions that satisfy the implementation (they are in L_i) but do not satisfy the specification (they are not in L_s). These are under-specified. If the intuitive abstraction one has in mind is one of these, the specification must be changed. Too many values are equal in speceq . This might be corrected by removing or rewriting an axiom. If the intuitive abstraction is not in L_i then both the specification and the implementation must be changed.

When L_i is completely contained in, but not equal to L_s, every semantic interpretation that satisfies the implementation satisfies the specification. In this case $SL = L_i$. Every data abstraction in SL is over-specified. If the intuitive abstraction one has in mind is in SL, both specification and implementation are correct. If the intuitive abstraction is in L_s but not L_i, then only the implementation needs to be changed. Too many values are equal in conceq. This might be corrected by adding more tests in the code or by adding concrete variables to the representation of some type(s). The new variables would be used to discriminate between values of that type. If the intuitive abstraction is not in L_s then both specification and implementation must be changed.

When L_s and L_i are equal every semantic interpretation that satisfies the specification satisfies the implementation, and vice versa: $SL = L_s = L_i$. The semantic interpretation speceq is also the semantic interpretation conceq; it is neither over-specified nor over-implemented. Every other interpretation in SL is both over-specified and over-implemented. If the intuitive abstraction one has in mind satisfies either the specification or the implementation then it satisfies the other. If it does not satisfy one then it does not satisfy the other. This case is unlikely to occur often in practice, we feel, because the techniques of specification and implementation are so different. Specification is more algebraic in flavor, and makes some distinctions easier (and some distinctions harder) to express than in implementations.

Finally, when L_s and L_i are not equal or related by containment, some semantic interpretations satisfy the specification and not the implementation, some interpretations satisfy the implementation and not the specification. Some interpretations, all those in SL, satisfy both, and are both over-specified and over-implemented. The example in Fig. 5.1 shows this case for the List data abstraction, whose specification and implementation appear in Fig.s 3.1 and 4.1, respectively. The semantic interpretation speceq is not in L_i. If this were one's intended abstraction the implementation would need to be changed. Similarly, the specification would need to be changed if the interpretaion conceq were one's intended abstraction.



A = EmptyL
B = Makelist(One)
C = Conc(Makelist(One),Makelist(One))
D = Conc(Conc(Makelist(One),Makelist(One)),Makelist(One))

Figure 5.1 Lattices L_s , L_i and SL for List

5.2. Maintenance Issues

As we stated before, under-specification and under-implementation are instances of incorrect specification and implementation, respectively. Some values of the data abstraction that can be distinguished intuitively are not distinguished by the specification or implementation. Over-specification and over-implementation are not incorrect. The smaller the sublattice SL is, the greater the probability is that the intuitive data abstraction has been under-specified, under-implemented or both. On the other hand, the larger the sublattice SL is, the greater the probability is that the specification and implementation are correct.

In software maintenance one is often required to modify specifications and implementations to reflect new uses of the software. Robust software survives many new applications with little or no required modification. For data abstractions this quality of robustness is reflected in the size of the sublattice SL . The larger the sublattice is, the more interpretations are allowed. Since short specifications (i.e., few axioms) tend to have large lattices, over-specification is encouraged by robustness and clarity. Small implementations (i.e., few concrete variables and few lines of code), on the other hand, tend to have small lattices: each distinction in values of the abstraction "costs" something, existence of a new variable or existence of a new test. For this reason, over-implementation may be discouraged.

It is our view that specifications and implementations reflect the depth of understanding of their author. Two different interpretations of a specification or an implementation may be equally acceptable if their differences are not important to (that is, not intended by) the author or the user. In such cases the best description of one's intentions is that collection of interpretations that are acceptable. The lattice structure provides a handy tool for describing such a collection, and for making further discriminations when they are needed.

6. References

[ADJ 77]
J. A. Goguen, J. W. Thatcher, E. G. Wagner and J. B. Wright
"An initial algebra approach to the specification, correctness, and implementation of abstract data types"
IBM RC 6487 April 1977

[Birkhoff 67]
Garrett Birkhoff
Lattice Theory
AMS Colloq. Pub.s v 25 (3rd ed.)
1967

[Birkhoff & Lipson 70]
G. Birkhoff and J. Lipson
"Heterogeneous Algebras"
J. Combinatorial Theory v. 8 pp. 115-133
1970

[Dahl et al. 70]
O. -J. Dahl, B. Myhrhaug and K. Nygaard
The SIMULA 67 Common Base Language
Pub. S-22, Norwegian Computing Center, Oslo
1970

[Gannon & Rosenberg 78]
J. D. Gannon and J. Rosenberg
"Data abstraction facilities in SIMPL-D"
Proc. ACM/NBS 17th Annual Tech. Symp. pp. 55-63
1978

[Giarratana et al. 76]
V. Giarratana, F. Gimona and U. Montanari
"Observability concepts in abstract data type specifications"
Math. Foun. Comp. Sci. 76 pp 576-587
1976

[Grätzer 78]
George Grätzer
General Lattice Theory
Academic Press
1978

[Guttag 77]
John V. Guttag
"Abstract data types and the development of data structures"
CACM v. 20 n. 6 pp. 396-404
June, 1977

[Hamlet 77]
Richard G. Hamlet
"Testing programs with the aid of a compiler"
IEEE TSE v SE-3 pp 279-290
1977

[Hamlet 78]
Richard G. Hamlet
"Structured computability"
LN-6 Dept. of Computer Science, Univ. of Maryland
1978

[Hamlet et al. 79]
R. Hamlet, J. Gannon, M. Ardis and P. McMullin
"Testing data abstractions through their implementations"
Univ. of Md. Comp. Sci. TR-761
May 1979

[Hoare 72]
C. A. Hoare
"Proof of Correctness of Data Representations"
Acta Informatica v. 1 n. 4 pp. 271-281
1972

[Kleene 52]
Stephen C. Kleene
Introduction to Metamathematics
Van Nostrand
1952

[Linger, Mills & Witt 79]
R. C. Linger, Harlan Mills and B. I. Witt
Structured Programming Theory and Practice
Addison-Wesley
1979

[Liskov et al. 77]
B. H. Liskov, A. Snyder, R. Atkinson and C. Schaffert
"Abstraction mechanisms in CLU"
CACM v. 20 n. 8 pp. 564-576
August, 1977

[Polajnar 78]
Jernej Polajnar
"An algebraic view of protection and extendability in abstract data types"
Ph. D. diss. USC September 1978

[van Wijngaarden et al. 76]
van Wijngaarden, Mailloux, Peck, Koster, Sintzoff, Lindsey, Meertens and Fisker
Revised Report on the Algorithmic Language Algol 68
Springer Verlag, 1976, Section 7.4

[Wulf, London & Shaw 76]
William A. Wulf, Ralph L. London and Mary Shaw
"An introduction to the construction and verification of Alphard programs"
IEEE Trans. Soft Engin. v. 2 n. 4 pp. 253-264, December, 1976

[Zilles 75]
S. N. Zilles
"Algebraic specification of data types"
MIT CSG Memo 119 pp. 1-12
July, 1975

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) Unclassified

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 79-1154	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE STRUCTURE OF SPECIFICATIONS & IMPLEMENTATIONS OF DATA ABSTRACTIONS		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER TR-801
7. AUTHOR(s) Mark A. Ardis and Richard G. Hamlet		8. CONTRACT OR GRANT NUMBER(s) AFOSR 77-3181
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Maryland Department of Computer Science College Park, Maryland 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB, Washington, DC 20332		12. REPORT DATE September 1979
		13. NUMBER OF PAGES 51
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A data abstraction is a collection of sets together with a collection of functions. An intuitive abstraction is unconnected with formalism: the sets and functions are supposed to be known ab initio. Formal ideas enter when the abstraction is (i) implemented, a conventional program written to carry out the operations on acutal data; and (ii) specified, a mathematical characterization given to precisely describe its sets and functions. The intuitive abstraction, an implementation, and a specification share a syntax that names the sets and functions, and gives the function domains and ranges

DD FORM 1 JAN 73 1473

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract continued.

(as set names). The central question for any particular example of syntax is whether the semantics of the three ideas correspons: does the collection of objects and operations a human being was thinking of behave in the way the implementation's data and procedures behave? Do the mathematical entities behave as imagined? The questions can never be answered precisely, because the intuitive abstraction is imprecise. On the other hand, precise compariso of specification and implementation is possible.

This paper presents an algebraic comparison of specifications with implementations. It is shown that these abstractions always overlap, and hav a common (lattice) structure that is valuable in understanding the modification of code or specification. However, in dealing with the precise entities subject to formal analysis, we must not lose sight of the intuition behind them. Therefore, our definitions are framed in terms of the intuitive abstraction a person attempted to specify or implement, and we refer the algebraic ideas to this standard whenever possible.

Section 1 presents the intuitive ideas of an abstraction, its implementation, and specification. The ideas are essentially those of (Hoare 72) and (Guttag 77). Section 2 gives the common formalism to be used the constrant work algebra. In Sections 3 and 4, this is applied to specification and implementation. Section 5 explores the overlap between the ideas, and suggests that the precise connection can shed light on the imprecise one that is really of interest: the intuitive abstraction in a person's mind.